Programming Project #11

Assignment Overview
This is round three of our combination data structure, `MapSet`, which combines a map and a set. We change the specifications again so that now you will use a singly linked list as the underlying data structure. It is due 04/16, Monday, before midnight on Mimir. Project is worth 65 points (6.5% of your overall grade).

Background
As this is the third time through on the MapSet, I assume you get the specifications of the individual methods. However, the update here is focused on using a linked list instead of a dynamic array. The overall idea remains the same:

- this is a templated class, two template types of key and value
- the list is always in sorted order according to the key of each Node
- the list can grow dynamically in size over the course of the run

The spec below is different. You already know how the `MapSet` works. This is more about the pitfalls that are coming your way when using a linked list. **You should really, really read this**. It is advice that will help you!

Differences

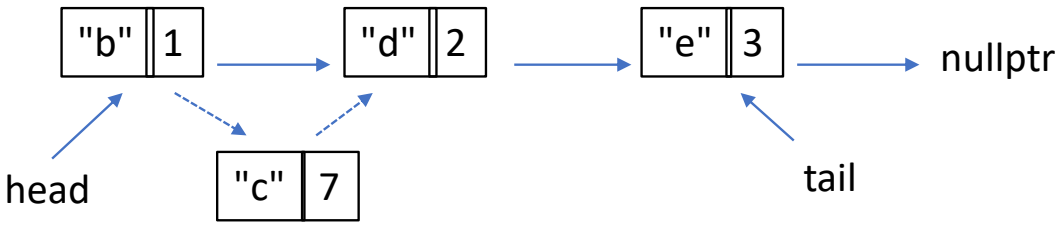
1. **No iterators, no pointer arithmetic, no algorithms**. In a singly linked list of our own construction, there are no iterators. C++ has the ability to add iterators to a class, but it is a bit beyond us. We got around iterators in arrays by using pointer arithmetic. Because an array is a **single, contiguous, piece of memory**, pointer arithmetic allows us to move to the next element of the array, just like an iterator. However, there are no iterators in our singly linked list. Likewise, pointer arithmetic doesn't apply. Each `Node` is individually and separately created at a memory location chosen by the OS. There is no relationship between the memory addresses and the order of linked elements. We can only use the `next` pointers of each `Node` to put them in some order.

   Without iterators/pointer_arithmetic, you cannot use any of the STL algorithms. None. You can't sort, you can't lower_bound, you can't copy. Nothing.

2. **find_key is sequential**. The essential method used to find an element in our linked list can no longer use `lower_bound`. Further, even though the elements in the `MapSet` must be in sorted order, we cannot really do a binary search even if we construct that search ourselves. This is a fundamental limitation of a singly linked list: you can move only one direction, forward, through a list. A doubly linked list can move forward and backwards and that is a necessary requirement of a binary search

   This means that `find_key` is going to be a **sequential** search. That is, we must start at the beginning of the list and move forward until we find what we are looking for. Shame as it is sorted, but that is the penalty incurred with single linking.

3. **add/remove and find_key**. The fact that this is a single linked list causes us some more headaches as well. Consider the diagram below.



Let's try to "find" where to insert {"c",7}. If `find_key`, using sequential search, uses the behavior we found in `lower_bound`, then the return location would point to {"d", 2}, the first element greater than the element being inserted. That is a real problem! We can change the newly made node containing {"c", 7} to point to {"d", 2}, but we cannot go backwards to update that {"b",1} points to the new node {"c", 7}.

We could change `find_key` to imitate `upper_bound` (look it up), which finds the less than or equal value. In this case, it would return {"b", 2}, and we could do the proper updates.

However, with either `lower_bound or upper_bound` behavior, what about remove? If we try to remove {"d", 2}, then either algorithmic approach to `find_key` would point to the {"d",2} Node. We could not update that the {"b", 1} should point to the {"e", 3} node (cutting out the {"d",2} Node as required) because we have no way to go backwards.

You have two choices here:

a. You can use either `upper_bound` or `lower_bound` behavior in `find_key`, but then update add/remove to search sequentially for the pointer "just behind". That is, use `find_key` to "find" the Node, then walk forward (again) from the beginning up to the Node "just behind" the pointer that `find_key` returned.
b. You could update `find_key` to return **two pointers** (as a pair for example). One of the pointers is the `lower_bound/upper_bound` pointer and the other is the "trailer" pointer, the one just behind.
c. There is a third, bad choid. You could just skip `find_key` and put the behavior everywhere you need it, but you need it in a lot of places and you are going to screw up that way. Not modular at all!

I did ***b*** above, but I will leave the header ambiguous in the return type and you can do as you wish.

4. **Another note about add**: This is a hard error to dig out so I'll warn you now. `add` takes a `Node` parameter, the `Node` you are trying to add. You will be tempted to simply link that passed-in parameter into your existing list***. Don't!*** If you do you'll eventually get a very strange RunTime (not Compile) error coming from the destructor. It says something to the effect that "`delete pointer being freed was not allocated`". When the destructor is called (look at the example code) it deletes each `Node` in turn. delete assumes that each `Node` was a result of a call to `new`, that is dynamically allocated. If you didn't allocate the `Node` with `new`, you cannot `delete` it. The parameter `Node` you passed was a declared variable, not a result of `new`. You must create a `new Node` using the first and second of the passed parameter to avoid this error.

5. **another weird error**. While we are on the subject, here is another that is hard to dig out. "`pointer being freed was not allocated`" . This is again a RunTime (not Compiler) error. It basically means that you tried to `delete` a `Node` twice. This can happen if you delete something in a method (somewhere) and leave it hooked into in the list. When the destructor is called, it tries to delete it again. Can't do that.

6. **nullptr and &&.** You are going to segfault a lot. Why? Imagine, however you implement `find_key`, that what you are looking for is not found. At that point your return value will be nullptr (off the end of the list,

so what you were seeking was not there). If you have something like the following, you are guaranteed to fault:

```
...
auto itr = find_key("z");
if (itr->first == key){
…
```

What happens if `find_key` returns nullptr? When you try to dereference `itr->first`, it segfaults. You cannot dereference a nullptr. But you can easily protect yourself:

```
...
auto itr = find_key("z");
if (itr != nullptr && itr->first == key){
…
```

This works because && is sequential. It evaluates each clause in order, and the first clause that is false halts the evaluation of the remaining clauses ("short circuiting", week 1 video!!!). If the variable is nullptr, then none of the remaining clauses execute. You should use that!!!

7. **Mimir segfaults**. Mimir has an interesting response on tests to segfaults. It doesn't explicitly say there is a fault, it just shows no output for that test and quits.

8. **linked list iteration**. This is the for loop to iterate through a linked list

```
…
for (auto itr = head_; itr != nullptr; itr = itr->next){
…
```

It isn't `++itr` (remember, no pointer arithmetic), it's whatever the next pointer points to. Remember that. Also, `++itr` will compile fine. It will segfault when you run!

9. **all the cases**. You have to enumerate all the possibilities and deal with them in your code. For example, the add function has at least 4 cases:
    a. the Node is already there
    b. We're adding a node and:
        i. it goes at the front
        ii. it goes at the back
        iii. it goes somewhere in the middle

    Each might have its own needs and conditions. Look at all the cases!

10. **Use the debugger**. Look, segfaulting sucks but it happens a lot, so get used to it. However, the only truly effective way to figure out where a segfault occurred is by using the debugger. If you ask for help by saying "Why is my code segfaulting", the only reasonable answer is "Where does the debugger tell you it is segfaulting". No one knows until you answer that question. You only need a few things:
    a. g++ … -g … use the -g to get debugging information in your executable
    b. gdb a.out
    c. run
    d. up/down to find code that looks familiar. You have to move up to get out of library code you didn't write until you find code you wrote
    e. list shows code around where the fault occurred
    f. print variable allows you to print a variable (find that null pointer, address 0x0000…).

11. **Use the example code.** As before, feel free to use the example code from the course. You have to modify it of course, but it can be very helpful. I will not accuse anyone of cheating if you are using the course example code!

Class Node
Here is the header part of Node. Not much difference except for the `next` pointer .

```
template<typename K, typename V>
struct Node {
  K first;
  V second;
  Node *next = nullptr;

  Node() = default;
  Node(K,V);
  bool operator<(const Node&) const;
  bool operator==(const Node&) const;
  friend ostream& operator<<(ostream &out, const Node &n){
     // YOUR CODE HERE!
  }
};
```

Much of this does not change I would think from your previous work.

Class MapSet
Here's the updated MapSet

```
template<typename K, typename V>
class MapSet{
 private:
  Node<K,V>* head_ = nullptr;
  Node<K,V>* tail_ = nullptr;
  size_t sz_ = 0;
  SOMETYPE find_key(K);

 public:
  MapSet()=default;
  MapSet(initializer_list< Node<K,V> >);
  MapSet (const MapSet&);
  MapSet operator=(MapSet);
  ~MapSet();
  size_t size();
  bool remove (K);
  bool add(Node<K,V>);
  Node<K,V> get(K);
  bool update(K,V);
  int compare(MapSet&);
  MapSet mapset_union (MapSet&);
  MapSet mapset_intersection(MapSet&);

  friend ostream& operator<<(ostream &out, const MapSet &ms){
     // YOUR CODE HERE
  }
};
```

Data Members

- `head_` is the pointer to the first `Node` (nullptr if the list is empty). `tail_` is not all that useful in this

project and I don't think dealing with it helps much. I think you can get away with not updating it. I don't believe anyting in testing depends on it though you might find it useful in your code. Up to you.

- `sz_` is the number of `Node`s in the list. You don't have to update it, but it makes many things easier if it accurately reflects the size of the list. The `size()` method is tested.
- `find_key` return type is your choice, as indicated above (but don't use SOMETYPE, that would be weird. Change it to what you need).

Methods

There isn't a lot different (in terms of input parameters and outputs) between 10 and 11. If you are careful, you can preserve a lot of what you did in 10

Two that might be different are `intersection` and `union`. If you used the algorithms, now you have to do that work yourself. Think carefully about how that might work. Done well, they are each a single, simple loop.

Requirements

We provide `proj11_mapset.h` as a skeleton that you must fill in. You submit to Mimir `proj11_mapset.h`

We will test your files using Mimir, as always.

Deliverables

`proj11/proj11_mapset.h`

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified directory and file name.

Assignment Notes